Inhalt

10 Strukturierte Datentypen - Felder

10.1 Deklaration und Definition von Feldern

Feld der Länge n $=_{df}$ n-Tupel über eine Menge von Objekten gleichen Typs

Felder sind strukturierte Datentypen, die für das Programm eine Einheit bilden und dabei Daten *gleichen* Typs zusammenfassen.

Analog den Variablen haben Felder einen **Namen** und einen **Typ**, aber **keinen Wert**. Außerdem haben sie eine **Länge**. Die Objekte, die durch ein Feld zusammengefasst werden, heißen **Feldkomponenten**. Sie besitzen einen **Wert**.
Felder werden wie folgt (statisch) deklariert:

Syntax: *Feldtyp Feldname* **[** *Länge* **] ;**

Vektor: **float vektor[3];**
Matrix: **int matrix[4][3];**

Feldtyp:
- *Typ* der Feldkomponenten, beliebiger Datentyp
- Der *Feldtyp* kann selbst ein *strukturierter Typ* sein.
  Dadurch können mehrdimensionale Felder vereinbart werden.
  (Matrix: Vektor mit m Komponenten, welche seinerseits wiederum jeweils aus einem Vektor mit n Komponenten gebildet wurden.)

Feldname:
- *C-Name*, mehrere Objekte **gleichen Typs** werden unter einem *Feldnamen* zusammengefasst.
- Der *Feldname* ist ein **Zeiger** und zeigt auf die Feldanfangsadresse im Speicher.

Länge:
- *int-Konstante*, gibt die Anzahl der Komponenten eines Feldes an.
- Sie muss zur Compilerzeit festgelegt sein und darf deshalb durch *keinen* Ausdruck dargestellt werden.
- Die Komponenten eines Feldes werden mit 0 beginnend bis *Länge* – 1 durchnummeriert.

Zugriff auf die Feldkomponenten durch den [ ]-Operator

Vektor:	**float vektor [3];**	=>	**vektor [0]**	**vektor [1]**	**vektor [2]**
Matrix:	**int matrix [4][3];**	=>	**matrix [0][0]**	**matrix [0][1]**	**matrix [0][2]**
			matrix [1][0]	**matrix [1][1]**	**matrix [1][2]**
			matrix [2][0]	**matrix [2][1]**	**matrix [2][2]**
			matrix [3][0]	**matrix [3][1]**	**matrix [3][2]**

Speicheranordnung
Im Speicher werden die Feldkomponenten **linear** angeordnet:

vektor[0]	**vektor[1]**	**vektor[2]**
Wert	*Wert*	*Wert*

matrix[0]			**matrix[1]**			
matrix [0][0]	**matrix [0][1]**	**matrix [0][2]**	**matrix [1][0]**	**matrix [1][1]**	**matrix [1][2]**	**. . .**
Wert	*Wert*	*Wert*	*Wert*	*Wert*	*Wert*	

Operationen
Operationen mit **Feldkomponenten** sind analog den **einfachen Variablen** erlaubt. Der Speicherbereich eines Feldes ist durch dessen Anfangsadresse (den Feldnamen), der Feldlänge und dem Datentyp seiner Komponenten eindeutig festgelegt. Beim Zugriff auf Feldkomponenten ist, ausgehend vom Feldanfang, intern eine **Adressrechnung** notwendig.

=> Statt **matrix[2][2] = matrix[2][2] * 2;** besser: **matrix[2][2] *= 2;**

Der folgende Programmausschnitt setzt die Komponenten eines Feldes auf 0:

```
# define LAENGE 10
int i;
float vektor[ LAENGE];
for( i = 0; i < LAENGE; i++) vektor[ i] = 0;
```

Ein Programm muss selbst absichern, dass beim Zugriff auf Feldkomponenten der Speicherbereich des Feldes nicht überschritten wird!

Bemerkung zu fencepost[1] - Fehlern
Betrachtet man lineare Felder in anderen Programmiersprachen, so stellt man fest:

Pascal	Untere und obere Grenze **müssen** festgelegt werden (**m..n**). Das Feld hat dann **n-m+1** Komponenten.
Basic	Untere Grenze ist mit 0 festgelegt. Obere Grenze **muss** festgelegt werden. Ein Feld der Länge **n** hat **n+1** Komponenten (**0..n**).
C	Untere Grenze ist mit 0 festgelegt. Obere Grenze **muss** festgelegt werden. Ein Feld der Länge **n** hat auch **n** Komponenten (**0..n-1**).

Nachteil
- Bereichsüberschreibungen werden nicht überprüft. Greift ein Programm auf eine nicht vorhandene „**n**. Komponente“ zu, so können Werte verloren gehen, die durch die Bereichsüberschreitung überspeichert werden.
  Im folgendem Beispiel wird einmal zuviel auf 0 gelöscht:

```
# define LAENGE 10
int i;
float vektor[ LAENGE];
for( i = 0; i <= LAENGE; i++) vektor[ i] = 0;
```

Vorteil
- Die Form der Schleifenanweisungen, insbesondere **for**-Anweisungen, sehen i.R. gleich aus. Damit werden sogenannte **fencepost-Fehler** (Zaunspfosten), auch **off-by-one-Fehler** (eins-daneben), verhindert. Diese treten häufig auf und sind schwer zu analysieren.

[1] Zaunspfosten-Fehler: Frage: Wie viel Zaunspfosten werden auf einer Strecke von 100 m Zaun mit 10 m Abstand benötigt?
Antwort: 100 / 10 = 10. => **falsch!**

10.2 C-Zeichenkettendeklaration

In C sind Zeichenketten lineare Felder vom Typ char, wobei die letzte Komponente mit dem NULL-Zeichen (´\0´) belegt wird.

```
char s[ 4];                          /* 3 Zeichen und \0 */
```

Direkte Möglichkeiten mit Zeichenketten zu arbeiten, gibt es in C nicht. Zur Verfügung stehen aber Funktionen der Standardbibliotheken.

< string.h >	Zeichenkettenoperationen
< ctype.h >	Funktionen zur Zeichenklassifikation
< stdlib.h >	Konvertierungsfunktionen

Diese Funktionen arbeiten nach zwei Methoden:
1. **Stringmethode:** Endeprüfung auf ´**\0**´, langsamere Methode.
2. **Puffermethode:** Arbeiten mit der Stringlänge, Speicher für Länge erforderlich.

nat.c

```
/*
 * Einlesen einer natuerlichen Zahl
 */

# include <stdio.h>
# include <ctype.h>       /* Zeichenkettenoperationen */
# include <string.h>         /* Zeichenklassifikation */
# include <stdlib.h>      /* Konvertierungsfunktionen */

# define LAENGE 4

int main()
{
  char s[ LAENGE];                   /* 3 Stellen und \0 */
  int p, i, Zahl;

  do
  {
    printf( "Dreistellige natuerliche Zahl: ");
    scanf( "%3s", s);
    p = 0;
    for( i = 0; i < strlen( s); i++)  /* in string.h */
      if( !isdigit(( int)s[ i])) p = 1;/* in ctype.h */
  }
  while( p);

  Zahl = atoi( s);                      /* in stdlib.h */
  printf( "\nDie eingelesene Zahl ist %d.\n", Zahl);

  return 0;
}
```

10.3 Explizite Anfangswertzuweisung bei Feldern

Syntax: *Feldtyp Feldname* **[** *Länge* **]** **=** *Wertemenge* **;**

Wertemenge:

- Geordnete Menge von *C-Konstanten* entsprechenden Typs.

```
# define LAENGE 5

int vektor0[ LAENGE] = { 1, 2, 3, 0, 0};   =>        1 2 3 0 0
int vektor1[ LAENGE] = { 1, 2, 3};         =>        1 2 3 0 0
int vektor2[] = { 1, 2, 3};                =>        1 2 3 u u
```

u unbestimmt, `Laenge` wird explizit auf 3 gesetzt

```
# define ZEILEN 3
# define SPALTEN 2

int m0[ ZEILEN][ SPALTEN]
= { { 1, 0}, { 3, 4}, {0, 0}};             =>      1 0 3 4 0 0
int m1[ ZEILEN][ SPALTEN]
= { { 1}, { 3, 4}};                        =>      1 0 3 4 0 0
```

```
# define STRLAENGE 9

char text0[ STRLAENGE]
   = { 'B', 'e', 'i', 's', 'p', 'i', 'e', 'l', '\0'};
                                                =>       Beispiel\0
char text1[ STRLAENGE] = "Beispiel";            =>       Beispiel\0
char text2[] = "Ball";                          =>           Ball\0
```

`Laenge` wird explizit auf 5 gesetzt

```
char text3[] = "ENDE";                          =>           ENDE\0

  text2[ 0] = 'F';                              =>           Fall\0
```

Speicher:

```
 66 101 105 115 112 105 101 108   0   0   0   0
 66 101 105 115 112 105 101 108   0   0   0   0
 70  97 108 108   0   0   0   0  69  78  68  69
  0   0   0   0   u   u   u   u   u   u   u   u
```

u unbestimmt

Anfangsadressen bei Feldern sind oft durch 4 teilbar! Es wird stets mit ´**\0**´ aufgefüllt.

zahl2.c

```
/*
 * Felder mit Zeichenkettenkonstanten
 */

# include <stdio.h>

int main()
{
  int n;

  const char hunderter[ 10][ 7] =
  { "", "Ein", "Zwei", "Drei", "Vier", "Fuenf",
    "Sechs", "Sieben", "Acht", "Neun"};

  const char zehner[ 10][ 9] =
  { "", "zehn", "zwanzig", "dreiszig", "vierzig",
    "fuenfzig", "sechzig", "siebzig", "achtzig",
    "neunzig"};

  const char einer[ 10][ 7] =
  { "", "ein", "zwei", "drei", "vier", "fuenf",
    "sechs", "sieben", "acht", "neun"};

  printf( "Eingabe einer Zahl zwischen 100 und 999: ");
  do
  {
    scanf( " %d", &n);
  } while(( n < 100) || ( n > 999));

  printf( hunderter[ n / 100]);          /* Hunderter */
  printf( "hundert");

  switch( n % 100)                            /* Rest */
  {
    case  1: printf( "eins"); break;
                              /* Unregelmeaszigkeiten */
    case 11: printf( "elf"); break;
    case 12: printf( "zwoelf"); break;
    case 16: printf( "sechzehn"); break;
    case 17: printf( "siebzehn"); break;
    default: printf( einer[ n % 10]);       /* Einer */
             if( n % 100 / 10 > 1) printf( "und");
             printf( zehner[ n % 100 / 10]);/* Zehner*/
  }

  printf( "\n");

  return 0;
}
```

10.4 Zeiger und Felder

Zeiger können als Feldkomponenten Verwendung finden, aber auch auf Felder zeigen.

=> Im Zusammenhang mit Feldern kann man vereinbaren:

Feld von Zeigern:	**int * v [10] ;**
Zeiger auf Feld:	**int (* v) [10] ;**

10.4.1 Zeigerarithmetik

Da Zeiger stets typgebunden sind, wird das „Rechnen" mit Zeigern in Verbindung mit Feldern möglich.

Insbesondere ist der *Feldname* selbst eine Adresse, die Anfangsadresse des Feldes.

=> **Jeder *Feldname* ist ein Zeiger.**

```
float v[ 4], *z;

z = v;          /* z zeigt auf die 0. Komponente von v */
z++;      /* z zeigt auf die nächste Komponente von v */
```

10.4.2 Zeiger statt Felder

Mit einem Feld kann man in C nur zwei Dinge anstellen:

- Einen Zeiger auf die 0. Komponente setzen und damit die **Speicheradresse** des Feldes festlegen: *Feldname*.
- Die **Länge** des zu reservierenden Speicherbereichs festlegen: *Feldtyp* und [*Laenge*].

Alle anderen Feldoperationen werden intern *nur* mit **Zeigern** durchgeführt, auch wenn der Operand in Indexschreibweise programmiert wurde.

=> **Jede Indexoperation entspricht einer Zeigeroperation.**

& v[0]	ist äquivalent mit	**v**
& v[i]	ist äquivalent mit	**v + i**
v[0]	ist äquivalent mit	*** v**
v[i]	ist äquivalent mit	*** (v + i)**

Felder sind als Parameter in Funktionen erlaubt. Es wird die Feldanfangsadresse übergeben, also ein **Zeiger auf ein Feld**. Felder können auch als Ergebnistyp einer Funktion verwendet werden. Dann wird ein **Zeiger auf ein Feld** zurückgegeben.
Das folgende Beispiel zeigt den Umgang mit Zeigern auf ein Feld als Parameter von Funktionen. Bei dem Zugriff auf einzelne Komponenten wird Zeigerarithmetik genutzt.

vekKompSumme.c

```
/*
 * Summation von Vektorkomponenten
 * mit Zeigern und Zeigerarithmetik
 */

# include <stdio.h>
# define LAENGE 3

                /* Funktionsdeklarationen für Felder */
                        /* mit Zeiger als Parameter */
void vektorEingabe( float *, unsigned int);
float vektorSumme( float *, unsigned int);

int main()
{
  float v[ LAENGE];

      /* Funktionsaufrufe, Uebergabe der Feldadresse */
  printf( "\nVektoreingabe:\n");
  vektorEingabe( v, LAENGE);

  printf( "\nVektorkomponentensumme: %f\n",
           vektorSumme( v, LAENGE));

  return 0;
}

                             /* Funktionsdefinitionen */
void vektorEingabe( float *vektor, unsigned int n)
{
  unsigned int i;
     /* Einlesen der Komponenten mit Zeigerarithmetik*/
  for( i = 0; i < n; i++)
  {
    printf( "[ %d] Komponente: ", i);
    scanf( "%f", vektor + i);
  }
  return;
}

float vektorSumme( float *vektor, unsigned int n)
{
  float *z, s = 0;
                            /* Zeiger als Laufvariable */
  for( z = vektor; z < vektor + n; z++) s += *z;

  return s;
}
```

10.4.3 Zeiger sind keine Felder

Die folgenden Beispiele sollen nochmals verdeutlichen, dass Zeiger keine Felder sind:

```
char *p = "Ball"; p[ 0] = 'F';        /* Segmentation Fault */
```

In diesem Fall ist **p** als **Zeiger** auf eine **Konstante** vereinbart. Bei der darauffolgenden Anweisung wird aber **p** als Adresse eines **Feldes** behandelt. In der Regel endet dieser Versuch mit einem **Segmentation Fault** (**coredump**).

```
char q[] = "Ball"; q = "Fall";              /* Systaxfehler */
```

Keine Probleme gibt es hingegen, wenn **q** auf ein Feld mit "Ball" als Wert zeigt. Hier wird automatisch dem Feld eine Konstante auf einem Speicherbereich mit 5 Komponenten zugewiesen. Es ist aber **syntaktisch falsch**, einem Feld außerhalb der Deklaration eine Zeichenkettenkonstante zuzuweisen. Jetzt muss man auf die einzelnen Komponenten direkt zugreifen.

```
char q[] = "Ball"; q[ 0] = 'F';                  /* richtig */
```

10.4.4 Synechdoche

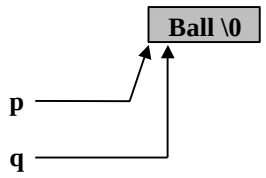
Eine Synechdoche ist ein literarisches Stilmittel, so etwas wie ein Vergleich oder ein Metapher.

Oxford English Dictionary:

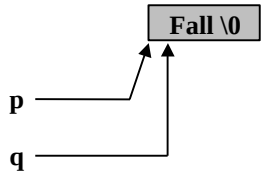
> „Ein umfassenderer Begriff für einen weniger umfassenden oder umgekehrt; ein Ganzes für einen Teil oder ein Teil als Ganzes; eine Gattung für ein Spezies oder eine Spezies als Gattung."

Die Definition beschreibt den häufig vorkommenden C-Fehler, dass ein Zeiger mit den adressierten Daten verwechselt wird.

```
char *p, q[] = "Ball";
p = q;
```




```
p[ 0] = 'F';
```



=> Beim Kopieren eines Zeigers wird das Objekt, welches adressiert wird, nicht kopiert.

10.5 Ein komplexes Beispiel für Felder

Mittelwertberechnung beliebig vieler Zahlen, in mehreren Modulen, kommentiert.

Algorithmus

1. Eingabe eines Vektors
2. Berechnung des Mittelwertes der Vektorkomponenten
3. Ausgabe des Vektors und seines Mittelwertes

Datentypen

Typ der Vektorkomponenten: ***float***
Anzahl der Vektorkomponenten: ***unsigned int***

2 Module (Schnittstellenfestlegung)

mwert.c

Hauptprogramm	***int main()***
Mittelwertberechnung	***float mittelwert(float *, unsigned int);***
Mittelwertausgabe	***void mittelwertAusgabe(float *, unsigned int);***

vektorio.h, ***vektorio.c***

Vektoreingabe	***unsigned int vektorEingabe(float *, unsigned int);***
Vektorausgabe	***void vektorAusgabe(float *, unsigned int);***
Vektorkomponentensumme	***float vektorSumme(float *, unsigned int);***

float ** als Parameter einer Funktionsvereinbarung erwartet eine Adresse als Argument, welche auf einen Wert vom Typ ***float zeigt. Hier ist es die Anfangsadresse eines Feldes vom Typ ***float***. Die Anzahl der Komponenten und damit die Feldlänge muss mit der Feldvereinbarung festgelegt und vom Programm überwacht werden.
Das Hauptprogramm mit der Funktion ***int main()*** befindet sich im Modul ***mwert.c***. Hier sind die Funktionen zur Berechnung des Mittelwertes und dessen Ausgabe zusammengefasst, also die Funktionen, die problemspezifisch sind.

mwert.c

```
/*
 * Mittelwertberechnung
 */

# include <stdio.h>
# include "vektorio.h"
# define MAXZAHL 10                /* Makrodefinition */

/*
 * Funktion 'mittelwert'
 * Berechnung des Mittelwertes eines Vektors
 * Parameter:   Zeiger auf einen Vektor
 *              Laenge des Vektors
 * Rueckgabe:   Mittelwert
 */
float mittelwert( float *, unsigned int);
```

```
/*
 * Funktion 'mittelwertAusgabe'
 * Ausgabe des Mittelwertes eines Vektors
 * Parameter:   Zeiger auf einen Vektor
 *              Laenge des Vektors
 */
void mittelwertAusgabe( float *, unsigned int);


int main()
{
  unsigned int Anzahl;  float vektor[ MAXZAHL];

  while(( Anzahl = vektorEingabe( vektor, MAXZAHL)) > 0)
    mittelwertAusgabe( vektor, Anzahl);

  return 0;
}


/*
 * Funktion 'mittelwert'
 */
float mittelwert( float *v, unsigned int n)
{
  if( n <= 0)
  {
    printf( "FEHLER: Es sind keine Werte vorhanden!");
    return 0;
  }
  return vektorSumme( v, n) / n;
}

/*
 * Funktion 'mittelwertAusgabe'
 */


void mittelwertAusgabe( float *v, unsigned int n)
{
  printf( "\n\nDas Mittel der %d Zahlen\n", n);
  vektorAusgabe( v, n);
  printf( "\nist %7.2f.\n\n", mittelwert( v, n));

  return;
}
```

Vektoroperationen sind für viele Anwendungen nützlich und sollen deshalb wiederverwendbar programmiert werden. Sie werden in einem anderen Modul zusammengefasst. Der dazugehörige Header beinhaltet alle Deklarationen der Vektorfunktionen, einschließlich einer Kommentierung. Diese dient Programmierern als Informationsquelle für die Art und Weise der Einbindung der Funktionen in andere Programme.

vektorio.h

```
/*
 * Vektoroperationen, Header fuer vektorio.c
 */

/*
 * Funktion 'vektorEingabe'
 * Eingabe eines Vektors
 * Parameter:   Zeiger auf den Vektor
 *              maximale Laenge des Vektors
 * Rueckgabe:   Laenge des Vektors
 */
unsigned int vektorEingabe( float *, unsigned int);

/*
 * Funktion 'vektorAusgabe'
 * Ausgabe eines Wertefeldes
 * Parameter:   Zeiger auf den Vektor
 *              Laenge des Vektors
 * Rueckgabe:   Anzahl der ausgegebenen
 *              Vektorkomponenten
 */
unsigned int vektorAusgabe( float *, unsigned int);

/*
 * Funktion 'vektorSumme'
 * Summe der Vektorkomponenten
 * Parameter:   Zeiger auf den Vektor
 *              Laenge des Vektors
 * Rueckgabe:   Summe der Vektorkomponenten
 */
float vektorSumme( float *, unsigned int);
```

Die Implementierung der Funktionen erfolgt in einer anderen Datei. Diese kann dann auch als Bibliothek weitergegeben werden. Hier ist eine Kommentierung angebracht, die bei der Wartung des Programms von Nutzen ist.

vektorio.c

```
/*
 * Vektoroperationen
 * Header: vektorio.h
 */

# include <stdio.h>
# include "vektorio.h"

/*
 * Funktion 'vektorEingabe'
 */
unsigned int vektorEingabe
  ( float *v, unsigned int max)
{
  float Wert; unsigned int i = 0;
```

```
  printf( "Geben Sie Zahlen ein und schliessen Sie ");
  printf( "die Eingabe mit ^d ab!\n");
  printf( "Oder: Beenden Sie sofort mit ^d!\n");

  while(( i < max) && (scanf( "%f", &Wert) != EOF))
    v[ i++] = Wert;

  return i;
}


/*
 * Funktion 'vektorAusgabe'
 */
unsigned int vektorAusgabe( float *v, unsigned int n)
{
  unsigned int i = 0;
  while( i < n) printf( "%11.2f\n", v[ i++]);
  return i;
}


/*
 * Funktion 'vektorSumme'
 */
float vektorSumme( float *v, unsigned int n)
{
  unsigned int i = 0; float Summe = 0;
  while( i < n) Summe += v[ i++];
  return Summe;
}
```

Das Makefile ***makefile*** gestattet eine unkomplizierte Erstellung eines ausführbaren Programms.

makefile

```
# makefile fuer mwert

CC = gcc
CFLAGS = -ansi -Wall -O -c
OBJ = vektorio.o mwert.o

mwert: $(OBJ)
   $(CC) $(OBJ) -o mwert

mwert.o: vektorio.h mwert.c
   $(CC) $(CFLAGS) mwert.c

vektorio.o: vektorio.h vektorio.c
   $(CC) $(CFLAGS) vektorio.c

clean:
   rm -f $(OBJ)
```